**CS2106 Introduction to Operating Systems**

**Lab 3**

**Answer Book**

Please read the instructions in the main lab sheet before completing this document. Submission deadline is **1 pm, Sunday 31 March 2024**. The folder will stay open slightly after this, but once the folder closes, **absolutely no submissions will be allowed.**

**Submission checklist:** A ZIP file called AxxxxxxY.zip, where AxxxxxxY is the student ID of the student submitting. The ZIP file should contain:

* Your answer book, properly renamed.
* Your barrier.c and barrier.h
* Your sum-par.c

|  |  |
| --- | --- |
| **Student 1** | |
| Name: Eunice Koh Shu Ning |  |
| Student ID (A0256986E): |  |
| Group (B08): |  |
| **Student 2** | |
| Name: |  |
| Student ID (AxxxxxxY): |  |
| Group (Bxx): |  |

**Part 1**

Question 1.1 (1 mark)

This pattern indicates that the time quantum for each child process is such that it allows each process to complete its entire execution before the next one gets a chance to run. The time quantum is long enough for a process to complete its task before being preempted.

Question 1.2 (1 mark)

The final counter value is 0 because each child process operates independently of the others, and they each have their own copy of the counter variable. In the parent process, where the final counter value is printed, the counter variable has not been modified by any of the child processes.

Question 1.3 (1 mark)

The output suggests that the time quantum allows each process to execute once before the time quantum expires, preempting the next child process in a round-robin fashion.

Question 1.4 (1 mark)

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

#define NUM\_CHILDREN 5

int main() {

int shmid;

key\_t key = IPC\_PRIVATE; // Private key for shared memory

int \*counter; // Pointer to shared memory

// Create shared memory segment

if ((shmid = shmget(key, sizeof(int), IPC\_CREAT | 0666)) < 0) {

perror("shmget");

exit(EXIT\_FAILURE);

}

// Attach shared memory

if ((counter = shmat(shmid, NULL, 0)) == (int \*)-1) {

perror("shmat");

exit(EXIT\_FAILURE);

}

\*counter = 0; // Initialize counter

pid\_t pid;

for (int i = 0; i < NUM\_CHILDREN; i++) {

pid = fork();

if (pid == 0) {

// Child process

printf("Child %d starts\n", i + 1);

// Simulate some work

for (int j = 0; j < 5; j++) {

(\*counter)++;

printf("Child %d increment counter %d\n", i + 1, \*counter);

fflush(stdout);

usleep(250000);

}

printf("Child %d finishes with counter %d\n", i + 1, \*counter);

exit(EXIT\_SUCCESS);

} else if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

}

// Parent process

for (int i = 0; i < NUM\_CHILDREN; i++) {

wait(NULL);

}

// Print the final value of the counter

printf("Final counter value: %d\n", \*counter);

// Detach shared memory

if (shmdt(counter) == -1) {

perror("shmdt");

exit(EXIT\_FAILURE);

}

// Remove shared memory segment

if (shmctl(shmid, IPC\_RMID, NULL) == -1) {

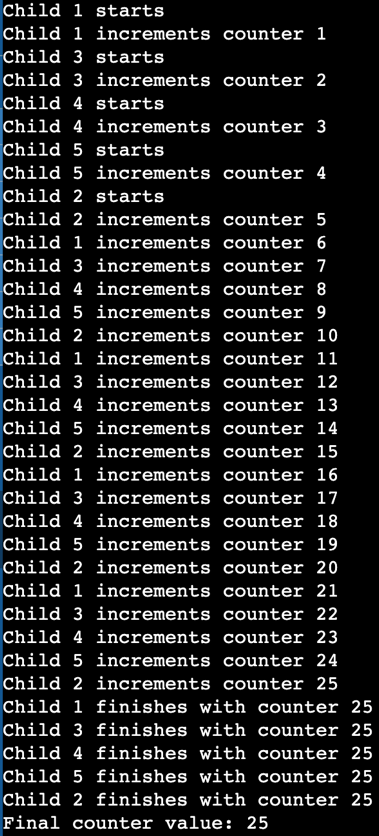
perror("shmctl");

exit(EXIT\_FAILURE);

}

return 0;

}

}

Question 1.5 (1 mark)

With more child processes trying to access and modify the shared counter variable, there's increased competition for this resource. This can lead to race conditions, where the final value of the counter may not be as expected due to unpredictable interleavings of increment operations by different processes.

Question 1.6 (1 mark)

Deadlocks can occur even in scenarios where locks are employed due to their lack of priority control. For instance, consider a situation where Child 3 requires a resource held by Child 2, but due to the absence of priority control, Child 2 is scheduled to acquire the lock last in the sequence, after Child 3. Child 3 may find itself unable to proceed because the resource it needs is held by Child 2, which is yet to acquire the lock. This situation can lead to a deadlock, where Child 3 is blocked, waiting for a resource held by Child 2.

Question 1.7 (1 mark)

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

#include <semaphore.h>

#include <fcntl.h>

#define NUM\_CHILDREN 5

int main() {

// Variables for shared memory

int \*counter, \*turn;

int shmid\_counter, shmid\_turn;

// Process ID

pid\_t pid;

// Create and attach shared memory for counter

shmid\_counter = shmget(IPC\_PRIVATE, sizeof(int), IPC\_CREAT | 0600);

counter = (int\*)shmat(shmid\_counter, NULL, 0);

// Create and attach shared memory for turn

shmid\_turn = shmget(IPC\_PRIVATE, sizeof(int), IPC\_CREAT | 0600);

turn = (int\*)shmat(shmid\_turn, NULL, 0);

// Initialize counter and turn to 0

\*counter = 0;

\*turn = 0;

// Fork child processes

for (int i = 0; i < NUM\_CHILDREN; i++) {

pid = fork();

if (pid == 0)

break; // Child process breaks out of loop

}

// Handle fork errors

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

// Child process logic

else if (pid == 0) {

// Wait for its turn

while (\*turn != getpid() % NUM\_CHILDREN);

// Child process

printf("Child %d starts\n", getpid() % NUM\_CHILDREN + 1);

// Simulate some work

for (int j = 0; j < 5; j++) {

(\*counter)++;

printf("Child %d increments counter %d\n", getpid() % NUM\_CHILDREN + 1, \*counter);

fflush(stdout);

usleep(250000);

}

// Finish

printf("Child %d finishes with counter %d\n", getpid() % NUM\_CHILDREN + 1, \*counter);

// Pass the turn to the next process

\*turn = (\*turn + 1) % NUM\_CHILDREN;

// Detach shared memory

shmdt(counter);

shmdt(turn);

exit(EXIT\_SUCCESS);

}

// Parent process waits for child processes to finish

for (int i = 0; i < NUM\_CHILDREN; i++) {

wait(NULL);

}

// Print the final value of the counter

printf("Final counter value: %d\n", \*counter);

// Detach shared memory

shmdt(counter);

shmdt(turn);

// Destroy shared memory segments

shmctl(shmid\_counter, IPC\_RMID, 0);

shmctl(shmid\_turn, IPC\_RMID, 0);

return 0;

}

This code creates shared memory for a counter and turn indicator. Child processes increment the counter in turns, simulating work. Each child waits its turn based on its PID. After finishing, the child passes the turn. The parent waits for all children to finish, prints the final counter value, and cleans up shared memory.

Question 1.8 (1 mark)

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

sem: This is a pointer to the semaphore variable to initialize.

pshared: This parameter indicates whether the semaphore should be shared between processes or threads. If pshared is 0, the semaphore is shared between threads of the same process. If pshared is non-zero, the semaphore is shared between processes.

value: This parameter specifies the initial value of the semaphore. It represents the number of processes or threads that can access the critical section protected by the semaphore simultaneously without blocking.

sem\_wait decrements the value of the semaphore, blocking the calling process or thread if the value is zero, indicating that the resource is currently being used by another process or thread. Once the resource becomes available, sem\_wait returns, and the calling process or thread can proceed to access the resource.

sem\_post increments the value of the semaphore, indicating that the calling process or thread has finished using the shared resource and is releasing it. If there are other processes or threads waiting on the semaphore, one of them will be unblocked and allowed to proceed.

Question 1.9 (1 mark)

In this program, a parent process creates a semaphore and initializes it with an initial value of 0 using sem\_init(). Then, it forks a child process. If the process is the parent (pid != 0), it waits for 1 second using sleep(1) and then increments the semaphore value using sem\_post().

The child process (pid == 0) waits for the semaphore value to become greater than 0 using sem\_wait(). Once the semaphore value is incremented by the parent, indicating that the parent has finished waiting, the child prints a message.

However, the issue arises due to the lack of shared memory between the parent and child processes. In this case, the semaphore sem is not shared between the parent and child processes. Each process has its own separate copy of the semaphore.

When the parent increments the semaphore using sem\_post(), it only increments its own copy of the semaphore, which doesn't affect the semaphore in the child process. Therefore, the child process remains blocked indefinitely in sem\_wait() because the semaphore value never becomes greater than 0 in its own context.

This situation leads to a deadlock or hang because the parent process expects the child to signal completion through the semaphore, but the child is waiting indefinitely for the semaphore value to change, which will never happen. As a result, the program stalls, and neither process progresses beyond their waiting states.

Question 1.10 (1 mark)

**Semaphore Initialization:**

An array of semaphores is created, with each semaphore associated with a specific child process. These semaphores are initialized with a value of 0.

**Child Process Execution:**

Each child process waits on its corresponding semaphore using sem\_wait(semaphores[child\_num]). This causes child processes to wait until their respective semaphore is posted by the previous child process.

After acquiring the semaphore, each child process executes its task and then posts to the semaphore associated with the next child process using sem\_post(semaphores[child\_num + 1]). If a child process is the last one (NUM\_CHILDREN - 1), it posts to the semaphore associated with the first child process to ensure the cycle continues.

**Parent Process Initialization:**

The parent process initializes the semaphore associated with the first child process (sem\_post(semaphores[0])). This allows the first child process to start its execution.

**Parent Process Handling:**

After all child processes have finished execution and terminated, the parent process prints the final value of the counter.

**Cleanup:**

After all processes have completed, semaphores are closed and unlinked, and shared memory segments are destroyed.

**Part 2**

Question 2.1 (1 mark)

void init\_barrier(int numproc) {

nproc = numproc;

// create a new shared memory segment for count

if ((shmid = shmget(IPC\_PRIVATE, sizeof(int), IPC\_CREAT | 0600)) == -1) {

perror("shmget for count");

exit(EXIT\_FAILURE);

}

// create a new shared memory segment for barrier

if ((shmid\_barrier = shmget(IPC\_PRIVATE, sizeof(sem\_t), IPC\_CREAT | 0600)) == -1) {

perror("shmget for barrier");

exit(EXIT\_FAILURE);

}

// create a new shared memory segment for mutex

if ((shmid\_mutex = shmget(IPC\_PRIVATE, sizeof(sem\_t), IPC\_CREAT | 0600)) == -1) {

perror("shmget for mutex");

exit(EXIT\_FAILURE);

}

// Shared memory attach operation for count

if ((count = (int \*)shmat(shmid, NULL, 0)) == (int \*)(-1)) {

perror("shmat for count");

exit(EXIT\_FAILURE);

}

// Shared memory attach operation for barrier

if ((barrier = (sem\_t \*)shmat(shmid\_barrier, NULL, 0)) == (sem\_t \*)(-1)) {

perror("shmat for barrier");

exit(EXIT\_FAILURE);

}

// Shared memory attach operation for mutex

if ((mutex = (sem\_t \*)shmat(shmid\_mutex, NULL, 0)) == (sem\_t \*)(-1)) {

perror("shmat for mutex");

exit(EXIT\_FAILURE);

}

\*count = 0;

// initialise the barrier semaphore

if (sem\_init(barrier, 1, 0) == -1) {

perror("sem\_init for barrier");

exit(EXIT\_FAILURE);

}

// initialise the mutex semaphore

if (sem\_init(mutex, 1, 1) == -1) {

perror("sem\_init for mutex");

exit(EXIT\_FAILURE);

}

}

The init\_barrier function initializes a barrier synchronization mechanism for a specified number of processes (numproc). It first sets the total number of processes (nproc) to the given input value. Then, it creates three shared memory segments using shmget: one for an integer count, and two for semaphores barrier and mutex. After that, it attaches these shared memory segments to the process address space using shmat. The count is initialized to 0, and the semaphores are initialized using sem\_init.

Question 2.2 (1 mark)

void reach\_barrier() {

sem\_wait(mutex);

(\*count)++;

sem\_post(mutex);

if (\*count == nproc) {

sem\_post(barrier);

} else {

sem\_wait(barrier);

sem\_post(barrier);

}

}

The function implements a barrier synchronization mechanism. Firstly, it locks a mutex semaphore using sem\_wait to ensure exclusive access to the shared count variable. Secondly, it increments the count variable by one, signaling that a process has reached the barrier. After releasing the mutex semaphore with sem\_post, it checks if all processes have arrived at the barrier by comparing count with the total number of processes nproc. If all processes have arrived (count equals nproc), it releases a barrier semaphore using sem\_post, allowing all waiting processes to proceed. Otherwise, it waits on the barrier semaphore using sem\_wait, ensuring synchronization among processes. Finally, it releases the barrier semaphore with sem\_post to maintain the correct synchronization state for subsequent executions.

**Part 3**

Question 3.1 (1 mark)

Firstly, the current timing setup begins measuring time after forking processes, failing to include the time spent on significant overheads involved in forking, such as context switching and memory allocation. Secondly, child processes may continue execution after the parent process has stopped the clock, leading to an incomplete measurement of their computational time. Finally, parallel computation introduces synchronization overheads, including barriers, which are not accounted for in the current timing mechanism. Combined, these factors contribute to an underestimation of the total time taken for the parallel computation, making the observed execution time lower than the actual time spent on parallel tasks.

(For grader only)

**Report: \_\_\_\_\_\_\_\_\_\_\_\_ / 13**

**Demo: \_\_\_\_\_\_\_\_\_\_\_\_\_ /7**

**TOTAL: \_\_\_\_\_\_\_\_\_\_\_\_\_/20**